Design Notes for New Telemetry Software

1 Overview

I'm presenting here a preliminary design for the telemetry processing software. This is intended as a starting point for our discussions, and the level of detail varies from section to section. Please feel free to e-mail me or the group with questions, suggestions, ideas, and criticisms.

This document is on the WWW at *http://sunland.gsfc.nasa.gov/~tcw/portable/Top.html*.

1.1 The Basic Telemetry Flow

```
Frame ----> Transfer frame ----> Packet       ----> Data      ----> CVT
source          sorter        re-assembler    unpacker
```

Transfer frames flow from the frame source, either an I&T front end or a ground station, to a transfer frame sorter. The sorter breaks the master channel into its virtual channels, and makes each virtual channel available in any conbination to downstream processes. The packet re-assembler reads transfer frames on a single virtual channel and re-assembles the source packets, which are made available to downstream processes. The data unpacker reads source packets, extracts individual telemetry values, and writes them into a current value table (CVT). (This is what the program `tlmClient` does currently.) Each unit here – the frame sorter, the packet re-assembler, and the unpacker – is either an executing process or a thread.

Downstream processes usually connect to upstream processes. All processes accept connections from any number of downstream processes. The sorter and re-assembler will share a common output module, which is actually what does the sorting and also does the archiving for most cases.

We additionally will construct an archiver program, a playback program, and a modified packet dumper. The archiver will accept frames or packets (It shouldn't care what it's getting.) and write them to disk. The playback program will read the archives and act as a data source exactly as if it were a front-end or ground station, frame sorter, or packet re-assembler. (Obviously the low-level interface to the various frame sources may be different.) The packet dump program will connect to this stream rather than using DBIF tags and channels as it does now. We may consider eliminating DBIF packet tags.

Where possible, I would like to take advantage of multi-threading in the design of the telemety system components.

1.2 The Common Output Module

The basic design of the output module is as follows: It has a single program interface through which it receives a data object an a sort key. The sort key is the VC id for transfer frames, the application id for packets. According to the key, the data object is sorted into output queues. A single object may be placed in any number of queues, even zero. If the entire object won't fit in a queue, no part of it is inserted in that particular queue. (Since the output module will use virtual memory for it's queues, this case never should arise.)

In a non-multi-threaded design, the output module returns a list of file descriptors to be tested for writeability (in a select). It also provides a function to be called when an output descriptor is ready to be written to.

In a multi-threaded design, each connection is handled by a thread which spends its time blocked on a write to the output, or blocked on a semaphore waiting for more data from the input thread.

Each output may be set up to do TCP/IP or ISIS communication.

1.3 The Transfer Frame Sorter

The transfer frame sorter receives the transfer frames from the frame source – a ground station, I&T front end, frame archive replay, and so on. It demultiplexes the virtual channels from the master channel and makes the individual virtual channels available to downstream packet re-assemblers and other processes.

The sorter unpacks from the transfer frame primary header the VC ID and the master channel sequence count. It checks the CRC status for the frame (attached by the frame source) and for master channel sequence errors, and keeps count of the number it finds. It optionally may drop any frames with bad CRC.

Through the memory management system, See Chapter 2 [Building Blocks], page 4 the sorter (and the re-assembler) also will perform frame archiving.

1.4 The Packet Re-Assembler

The packet re-assembler takes transfer frames on one virtual channel and re-assembles the source packets contained in the frames. If we want to look at packets from multiple virtual channels, we run one re-assembler for each VC in which we are interested.

When the re-assembler receives the first transfer frame, it unpacks the first header pointer from the frame header and begins extracting the packet. It continues chaining packets through transfer frames, checking that the virtual channel counter remains contiguous. If the VC sequence is broken, the re-assembler starts over. It either throws away or fills (and properly annotates) partial packets resulting from a VC sequence error.

The re-assembler should handle segmented packets, allocating temporary storage for partial packets as they are reconstructed. It should handle all CCSDS-approved segmentation schemes.

The re-assembler builds the packet annotation header using information from the transfer frame header, transfer frame annotation, and it's own observations.

1.5 The Data Unpacker

The data unpacker is a slightly modified tlmClient. Unpackers normally receive packets from one re-assembler only, but could receive from multiple re-assemblers, I suppose.

1.6 The Telemetry Flow Controller

The telemetry flow controller is the parent process of all other processes in the telemtry flow. It is an X program, which provides a graphical interface to the entire flow and to each process in it. STOL directives for flow control will be sent to this controller process. (I've made up a rough drawing of what the GUI might look like.)

The controller spawns each child at the end of a pipe or socketpair() connection. Children automatically configure their standard inputs as control inputs and the standard outputs as response/status outputs. The controller may kill its children to shut them down.

The controller offers an animated graphical display of the telemetry flow. There's an icon for each active process with lines among them indicating data flows. Users may drag an icon and drop it on another to form a connection. Alternatively, they may drop it on the canvas and use the mouse to connect it's data sources and sinks. Colors and flow-line annimation show data types and flows.

Clicking on an icon in the flow would bring up a status and configuration window for that process.

The controller also controls and displays status for the frame source.

I don't know what the default startup condition of child processes should be: Should they immediately begin listening for additional connections? Should they begin reporting status periodically?

Each program in the flow should come with a second program which provides the GUI specific to that program. The flow controller starts the program GUI when the user requests one. The GUIs should not be built in to telemetry handling programs for the sake of efficiency in those programs.

1.7 The Telemetry Archiver and Playback

Telemetry archiving normally will be done through a virtual memory buffer management package, documented elsewhere. However, an archiver program also should be provided to do archiving on remote systems if necessary. The telemetry archiver takes transfer frames or source packets and stores them in disk files.

The telemetry playback reads files created by the archiver and acts as a data source for the objects in the file: If the file contains transfer frames, a frame sorter can connect to the playback; if it contains source packets, a data unpacker can connect to the playback. It may be possible to allow a re-assembler to connect to a frame playback, also, depending on the details.

2 Building Blocks

The Frame Sorter and Packet Re-assembler programs are constructed from a collection of tasks and several other building blocks. Those other building blocks are presented here as programming objects.

I've tried to take an object-oriented approach to designing this software. I've divided it into somewhat interrelated object classes: pools, buffers, queues, and sorters.

Pools are virtual memory areas, memory mapped files. Buffers and queues are allocated from the pools.

Buffers hold a data objects like transfer frames or packets. Queues are FIFOs of buffer handles. Queues are intended to be associated with each output in what the introduction calls the Common Output Module. See Section 1.2 [Common Output Module], page 1

Note that pools and queues are different things. Many queues can reference buffers stored in the same pool. In practice, we are likely to have two active pools: one being archived, and one not. However, we will have many queues, each of which may reference buffers from both pools.

On the other hand, queues and outputs correspond one-for-one. Each output has a queue and every queue is associated with an output.

2.1 Memory Management

The frame sorter and the packet re-assembler may require large amounts of system memory to buffer data when receiveing high-rate telemetry. This certainly will be the case if the system is required to serve data to units which cannot keep up with the data rate; for example, a T1 line.

In the frame sorter and packet re-assembler, I want to use a simple, homemade memory management package that draws memory from the virtual memory system using the `mmap()` system call.

`mmap()` maps a disk file into virtual memory. This is the system call that loads shared libraries, and, on **Solaris 2.5**, underlies the `read()` and `write()` system calls. `mmap()` also gives a program access to swap disk space.

One advantage of using simplified memory management with `mmap()` over the `sbrk()` system call underlying `malloc()` is that memory allocated through `malloc()` and `sbrk()` cannot be returned to the system. Once a process grows with these calls, it cannot shrink. Using `munmap()`, programs acquiring memory with `mmap()` can give the memory back to the operating system when it no longer is needed.

Another advantage of using `mmap()` for our application is that it gives us built-in archiving. If we allocate transfer frame buffers, for example, with an `mmap()`-based system, we can simply leave on the disk the file we've mapped to obtain our virtual memory for those buffers.

2.2 pools of virtual memory

Pools are virtual memory areas from which buffers and queues are created; so, the pool object underlies both the buffer and queue objects.

Here are the principal data structures and methods for the pool object class:

The pool object maintains a list of memory pools. Each memory pool is an `mmap()`'d file. The size of each memory pool is set large enough that we don't have to create them often, but not so large that they occupy too much virtual memory. (I'd guess about 100MB would be about right.) We maintain a list of them so we can use as many as we need and free those we've finished with.

The pool object allocates buffers end-to-end sequentially through the list of individual memory pools. As one pool is exhausted, a new pool is created and added to the list. When the program is finished with all of the the buffers in a pool, the pool's memory is returned to the system.

The pool object supports two methods of accessing data in the pool buffers: sequential and non-sequential. If the buffers are to be accessed in the order they were created, and each buffer is to be accessed only once, the sequential method can be used. If the buffers are to be accessed in non-sequential order, or if each buffer may be accessed more than once, the non-sequential method must be used.

The pool object maintains a retrieve pointer for each memory pool and allows applications to retrieve buffers in the order they were allocated, but each buffer may be retrieved only once. If all the buffers in a memory pool have been retrieved, and the pool is no longer that from which new pools are allocated, the memory pool is removed from the list.

The pool also maintains reference counters for each memory pool. Since the pool allocate buffer method returns a buffer pointer, the buffer memory may be accessed directly (and in any order) without calling pool retrieve buffer. For use with multiple queues, we maintain a reference counter for each memory pool. If the counter falls to zero at anytime after the memory pool is no longer the one from which new buffers are being allocated, the memory pool may be removed from the list. Thus we can access the buffers in a pool in any order, but still only a number of times set.

2.2.1 pool data structures

Here is are the two principal data structures maintained by an instance of the pool class: The pool record type `PoolRec` (`struct pool_rec`) and the pool object type `Pool` (`struct pool_list`). The `Pool` structure manages a list of pool records. The `PoolRec` structure maintains state information on each pool in a given list.

2.2.1.1 Control record for each pool

```
typedef struct pool_rec PoolRec;
struct pool_rec
{
    char filename[FILENAME_MAX]; /* pool file name            */
    char *base; /* base address of pool      */
```

```
        char *get_mem; /* ptr to first free pool byte     */
        char *get_data; /* ptr to oldest unread byte     */
        char *end; /* highest address in pool + 1     */
        int  size; /* size of pool in bytes     */
        int  ref_count; /* count of references to this pool */
        PoolRec *next; /* link to next pool in list     */
        Pool *pool;                         /* pool list to which pool belongs  */
    };
```

2.2.1.2 Pool Control Structure (Pool List)

List of active pools. First pool in list has oldest unread data. Last pool is pointed to by "new_pool"; buffers are allocated from this pool. Once all data in pool has been read, pool may be dropped from list. If archiving of pool data is not in effect, pool file is deleted when pool is dropped from list.

```
    typedef struct pool_list *Pool;
    struct pool_list
    {
        char filename[FILENAME_MAX];        /* basename of pool files           */
        int file_sn;                        /* file serial number, from 0       */
        int flags;                          /* pool control flags               */
        int pool_size; /* size of all pools in the list    */
        int n_pools; /* number of pools in list     */
        PoolRec *pools; /* ptr to 1st pool in linked list   */
        PoolRec *new_pool; /* pool where data can be read     */
    };
```

2.2.1.3 Pool Buffers

The pool allocate buffer and retrieve buffer methods return pool buffer (`PoolBuf`) pointers.

```
    typedef struct pool_buffer PoolBuf;
    struct pool_buffer
    {
        char *p;                            /* pointer to buffer memory in pool */
        int len;                            /* length of buffer in bytes        */
        PoolRec *pr;                        /* memory pool where buffer lives   */
    };
```

2.2.2 pool create

The **pool create** method is the intialize method for the pool object and must be called before any other method. It goes ahead and creates an initial memory pool, so you get back a pointer to a control structure with a list of one pool. This method's declaration is:

`Pool pool_create(int pool_size, char *filename, int flags)`

where:

`pool_size`
: is the size of each pool. This number of bytes should be evenly divisible by the size of a virtual memory page for the system. This function probably should increase the pool_size transparently to the next highest value evenly divisible by the page size.

`filename`
: is the pathname of the file to be created to provide the disk space to back the virtual memory of the pool. A sequence number beginning with '`1`' is appended to the given name. If subsequent pools are needed, the suffix is incremented on subsequent file names. This argument is ignored if the `flags` arguments tells us to use swap space instead of a named file.

`flags`
: is used to turn on archiving for the pool and to tell us to draw the pool from swap rather than a named file.

A `Pool` handle is returned.

2.2.3 pool create pool

This method creates a backing file and does an `mmap()` to create a memory pool. This is how virtual memory is added to a process. The `PoolRec` structure itself, like the `Pool`, is created using good old `malloc()`, since we shouldn't need to many of them.

The **pool create pool** method is never called directly. It is called by the **pool allocate buffer** method whenever a new pool is needed to fulfill a request for a new buffer.

This method's declaration is:

```
PoolRec *pool_create_pool(Pool pool)
```

where:

`pool`
: is a pool handle.

A pointer to the newly created `PoolRec` is returned for convenience.

2.2.4 pool destroy pool

This method destroys a memory pool in a pool list. If the pool is not being archived and it has a file associated with it, this method also destroys the file.

It's declaration is:

```
PoolRec* pool_destroy_pool(Pool pool, PoolRec *pr)
```

where:

`pool`
: is a pool handle.

`pr`
: is the memory pool to destroy.

This method returns a pointer to the pool record for the next oldest memory pool from the one destroyed. If the given pool record is not recognized as part of the given pool, the given pool record pointer is returned. If there is no next oldest pool, NULL is returned.

2.2.5 pool allocate buffer

This method allocates an empty buffer from the next field of bytes available in the pool. It's declaration is:

```
char *pool_alloc_buffer(Pool pool, int *size)
```

where:

`pool` is a handle of the pool from which to allocate the buffer.

`size` is a pointer to the size in bytes of the buffer to allocate. The variable pointed to is set to the size of the buffer actually allocated. Since the certain alignment rules are enforced, this may be slightly larger than the buffer size requested.

This method returns a pointer to a pool buffer of the given size or NULL on errors.

2.2.6 pool retrieve buffer

This method retreives the next field of un-retreived bytes from a pool. It's declaration is:

```
char *pool_retrieve_buffer(Pool *pool, int *size)
```

where:

`pool` is a handle of a pool from which to retrieve the buffer.

`size` is a pointer to an integer containing the number of bytes to return. This value is changed to the number of bytes actually returned. It may be fewer than requested, if the end of a memory pool is reached in fulfilling the request.

The return value is a pointer to the requested field of bytes.

Each byte may be retreived only once. When all bytes in a memory pool have been retreived, the pool is destroyed, and it's memory returned to the system, provided that the pool is not at the last pool in the pool list.

2.2.7 pool adjust ref count

This method adjusts the reference count for a given memory pool. Reference counts are maintained on each pool in a pool list. They are used by applications which do non-sequential or multiple access to the buffers in the pool.

It's declaration is:

```
int pool_adj_ref_count(PoolRec *pool_rec, int adj)
```

where:

`pool_rec` is a pointer to an individual memory pool for which the reference count should be incremented.

`adj` is the number by which to adjust the reference count.

The return value is the new reference count for the given pool.

2.3 buffer queues

Buffer queues are FIFOs of buffer handles. The `BufQ` buffer queue class is a subclass of the Pool object class, and queues are created from Pool objects.

Here are the principal data structures and methods of the buffer queue object class:

2.3.1 queue data structures

```
typedef struct buffer_queue *BufQ;
struct buffer_queue
{
    Q_AccessControl control;
    Pool pool;
};
```

The `Q_AccessControl` field depends on what type of environment we are creating queues in. For a non-threaded environment, this is:

```
typedef struct queue_access_control Q_AccessControl;
struct queue_access_control
{
    int buf_count;
    int fd;
};
```

2.3.2 queue create

This method creates a new buffer queue (`BufQ`). It's declaration is:

`BufQ *queue_create(char *filename, int flags)`

where the `filename` and `flags` arguments are the same as for the Section 2.2.3 [pool create pool], page 7 pool object method.

This function returns a `BufQ` handle, which is a pointer.

2.3.3 queue destroy

This method destroys a buffer queue (`BufQ`). It's declaration is:

`void queue_destroy(BufQ* q)`

where `q` is the queue to be destroyed.

2.3.4 queue alloc buffer

The queue object "inherits" this entirely from it's superclass buffer. That being the case, just call the ⟨undefined⟩ [buffer alloc], page ⟨undefined⟩ method.

2.3.5 queue post buffer

This method posts a buffer to the end of given queue. It's declaration is:

`int queue_post_buffer(BufQ *q, PoolBuf *buf)`

where:

`q`	is the queue to which to add the buffer.
`buf`	is a pointer to the buffer to post.

This function returns the number of buffers in the queue.

2.3.6 queue get next buffer

This method retrieves the next buffer from the queue. It's declaration is:

`PoolBuf *queue_get_next_buf(BufQ *q, PoolBuf *buf)`

where:

`q`	is the queue from which to get a buffer.
`buf`	is a pointer to storage for the retrieved buffer handle.

This function returns the next buffer (`PoolBuf`) in the pool. It returns a NULL if the queue is empty.

2.3.7 queue control

intentionally left blank

2.4 pool management

The `Archiver` object manages as set of pools from which buffers may be allocated to hold a transfer frame or packet. The `Archiver` determines from which pool in a collection to allocate a buffer using the same sort key used by the buffer queue object. This is how we select which data items are placed in which archive. Note, however, that each may be archived only once.

The `Archiver` object class is a subclass of the Pool object class. Here are the principal data structures and methods for the pool buffer object class:

2.4.1 Archiver data structures

The `PoolBuf` if a data buffer. This is what queues and sorters use.

```
typedef struct pool_buffer PoolBuf;
struct pool_buffer
{
    char *p;                    /* pointer to actual buffer, pool memory    */
    int  len;                   /* bytes of memory in buffer                */
    int key;                    /* key associated with buffer data          */
    PoolRec *pool;              /* pool from which buffer allocated         */
    };
```

The `Archiver` is a handle to a collection of Pools from which buffers may be allocated. The list of Pools is indexed by the sort key, which is the virtual channel or packet ID.

```
    typedef struct buffer_pool BufPool, *Archiver;
    struct buffer_pool
    {
        int nkeys;                   /* number of keys on which to sort          */
        Pool *key_list;              /* list of pools, nkeys long                */
    };
```

2.4.2 archiver manage pool

This method places a Pool object under control of an Archiver. It's declaration is:

`Archiver arch_manage_pool(Archiver arch, Pool pool, int max_key, int *keys, int nkeys)`

where

`arch`	is a Archiver to which this pool is to be added. If this argument is NULL, a new Archiver is created and the given pool is used as the default pool, that is, all buffers are drawn from this pool if their key doesn't select any other. In this case, the `keys` and `nkeys` arguments are ignored and need not be specified.
`pool`	is the pool to add to the collection.
`max_key`	is the maximum key value. This must be less than or equal to the value given when the Archiver was created.
`keys`	is the list of keys which select this pool.
`nkeys`	is the number of keys in the list.

This method returns it's first argument with the new pool added, or a new Archiver containing only the given pool, if the first argument was NULL. It returns NULL on errors.

How does this stuff get used? Like this: First, create a pool with the `pool_create()` function see Section 2.2.2 [pool create], page 6, turning archiving on for the pool as appropriate. Then add the pool, with a list of keys, to a buffer object with `buffer_manage_pool()`. Whenever a buffer is allocated with `arch_alloc_buffer()`, the key determines from which pool the buffer is allocated, and, so, whether it is archived or not.

In practice we probably will never create more than two buffer pools: one which is being archived and one which is not.

I suppose it would be better, from an object-oriented perspective, to re-implement the create methods of the pool superclass, but this is simpler.

2.4.3 archiver alloc buffer

This method allocates a new buffer `size` bytes long from one of the pools in the collection `bp`. The `key` determines from which pool the buffer is allocated. It's declaration is:

`PoolBuf *buffer_alloc(BufPools bp, int key, int size)`

It returns a new `PoolBuf` buffer or NULL on errors.

2.4.4 buffer adj ref count

This method changes is inherited from the pool superclass. Simply call the Section 2.2.7 [pool adjust ref count], page 8 method.

2.5 queue management

The Sorter manages a collection of queues. The Sorter object is a subclass of the queue object.

Here are the principal data structures and methods of the sorter object class:

2.5.1 sorter data structures

```
typedef struct sorter_queues SorterQ, *Sorter;
struct sorter_queues
{
    int nkeys;                  /* number of keys in the key_list           */
    char *key_list;             /* sort list indexed by key value           */
    BufQ *q;                    /* buffer queue handle                      */
    SorterQ *next;              /* pointer to next queue managed by sorter  */
};
```

The `key_list` is a list of `nkeys` characters. Each entry in the list is indexed by a possible key value. If the entry's value is non-zero, the key selects this pool.

2.5.2 sorter manage queue

This method places a queue object under control of a sorter object. It's declaration is:

`SortQueues *sorter_manage_queue(SortQueues sq, BufQ *q, int max_key, int *keys, int nkeys)`

where

`sq`	is the sorter to which the queue is being added. If this argument is NULL, a new sorter object is created.
`q`	is the queue we're adding to the sorter.
`max_key`	is the maximum key value.
`keys`	is the list of key values for which this queue will be selected by Section 2.5.5 [sorter post buffer], page 13. If this is NULL, then this queue will be selected for all possible key values up to `max_key`.
`nkeys`	is the number of key value in the `keys` list. This parameter is ignored if `keys` is NULL.

This function returns a sort queue collection with the new queue added to it, or a new sorter object if the first argument is NULL. Note that the return value and first argument are never the same.

2.5.3 sorter alloc buffer

This is inherited from it's superclass. Just call ⟨undefined⟩ [buffer alloc], page ⟨undefined⟩ to get a buffer in all cases.

2.5.4 sorter post buffer

This method posts a buffer to zero or more queues managed by a sorter object. It's declaration is:

```
void sorter_post_buffer(Sorter sq, PoolBuf *buf, int key)
```

where

`sq`	is the sorter object.
`buf`	is the buffer to post.
`key`	is the key to use in selecting the queues to which to post the buffer.

This method returns the number of queues to which the buffer was added.

2.5.5 sorter post buffer

This method changes the keys which select a buffer queue (`BufQ`) managed by a sorter. It's declaration is:

```
int sorter_change_keys(Sorter sq, BufQ q, int *keys, int nkeys)
```

where:

`sq`	is the Sorter containing the queue for which we want to chage the keys.
`q`	is the queue for which we want to change the keys.
`keys`	is a list of key values. The absolute value of each item in the list is a key value. If an item is negative, the corresponding (positive) key will no longer select the given key.
`nkeys`	is the number of items in the `keys` list.

This method returns the number of valid key values processed. Normally this is `nkeys`, but may be less if some key values in the list exceeded the maximum key value for the queue. A zero return value may indicate that the given queue was not recognized as being managed by the given sorter.

3 Frame & Packet Handling

We need the frame sorter when the frame source is using TCP/IP so we can run packet re-assemblers on other computers in an effort to balance the workload.

If the frames are coming in over ISIS IP multicasts, we don't need a sorter: Each re-assembler can receive all frames and discard those they don't need. This way we get each frame on the network one time only, and the work in each re-assembler to input and reject frames on the wrong virtual channel is very little. If ISIS is not using true IP multicasting, then, once again, the sorter is needed to reduce the network traffic.

There are side benefits to such a design. In cases like I&T where the frame source is using TCP/IP, we need to sort the transfer frames by virtual channel so we can run some packet re-assemblers on other workstations to balance the compute load. But even so, we may re-assemble housekeeping packets on the workstation doing the frame sorting. Why do it in a separate process? This is especially true on Solaris where we have control over thread binding, and therefore, thread concurrency.

In a multi-threaded environment, the frame sorter and packet re-assembler both can be constructed from a collection of objects: input tasks, output tasks, packet assembly tasks, pool objects, sorter objects, and queue objects.

The packet re-assembler might looks like this:

```
  input --> queue --> pkt assy --> sorter --> queue --> output
```

Transfer frames enter an input. Input enqueues and possibly archives them. The packet assembly task reads frames out of the queue and constructs packets. It hands completed packets to a sorter which enqueues them for output tasks and possibly archives them. Each output task reads packets from its queue and sends packets out.

A control task handles new connections. It accepts a connection, read an initialization request, and creates a new output (if that's what was requested).

To build a frame sorter, drop the queue –> pkt assy. If we construct the tasks correctly, this should be no problem.

3.1 Threading vs. non-threading

Since ISIS presents a multi-threaded programming interface, it is convenient to design the software in a multi-threaded manner. Input and output tasks that do not use the ISIS protocols require very few changes to go from the ISIS task API to a native (POSIX) threads API.

Unfortunately, if a system has neither ISIS or a threads library, the software design changes to a larger degree. The data objects, like buffers, queues, and pools, are affected less than the execution objects like inputs and outputs.

There are four possible system configurations: have ISIS and have threads, have ISIS only, have threads only, and have neither.

ISIS & threads

TCP/IP input
: block on `read()` until frame received

enqueue
: increment threads semaphore (`sem_post()`)

dequeue
: block on semaphore (`sem_wait()`)

TCP/IP output
: block on `write()`

threads only

TCP/IP input
: same as ISIS & threads

enqueue
: same as ISIS & threads

dequeue
: same as ISIS & threads

TCP/IP output
: same as ISIS & threads

ISIS only

TCP/IP input
: loop on non-blocking `read()` and `isis_select` until frame received

enqueue
: increment in-queue count, possibly sent ISIS `t_sig()`

dequeue
: task-block on ISIS `t_wait()`, decrement in-queue count when `t_wait()` returns

TCP/IP output
: loop on non-blocking `write()` and `isis_select` until buffer written

neither

TCP/IP input
: non-blocking read, returning input file descriptor while partial frame is read, and posting whole frame when complete and returning zero.

enqueue
: add associated output file descriptor to fd_set.

dequeue
: if queue now empty, remove file descriptor from fd_set.

TCP/IP output
: non-blocking write, returning output file descriptor while buffer bytes remain to be written, and returning zero when whole buffer written.

3.2 TCP/IP frame input task

There are three flavors of input task, depending on whether native threads are available, and if not, whether ISIS threads are available or we don't have threads at all.

In general, the input task, reads a frame and posts it to a sorter or queue. Along the way it checks for master channel sequence errors. That's all.

3.2.1 native threads

The native thread receives as an argument a structure containing a file descriptor from which to read, and a handle for the sorter to which it is to post.

It executes a blocked `read()` in a loop until it has read the ITP and frame header up to the virtual channel ID. With the VCID as a key, it then allocates a buffer, and copies what it has read so far into that buffer. Then the task goes back to the read loop until the remainder of the frame has been read.

Once it has a complete frame, it checks the master channel sequence count and records any discontinuity. (The program maintains a count of master channel sequence errors.) Then it posts the buffer containing the frame to the sorter and loops back to begin reading the next frame.

3.2.2 ISIS threads

The best performance can be obtained from this thread (according to my reading of the ISIS documentation) by emulating the native threads task design. From that design, replace the blocked `read()` with an `isis_select()` followed immediately by a non-blocking `read()` in the same loop. The `isis_select()` allows us to task block waiting for input while allowing other ISIS tasks to run. Everything else remains the same.

To more closely emulate the non-threaded design, the input task could be created by `isis_input()`. In this case, everything else is as in the non-threaded design; however, the main loop mentioned is the `isis_mainloop()`.

3.2.3 no threads

If no threads package is available, the input task is a function. It receives the same argument as the thread in the designs above.

This function cannot, in general read a whole frame during one execution; it is called several times to read a complete frame. To support this, it keeps a static counter of how many bytes of the current frame it has read.

When the function is entered, it first checks to see if it is in the middle of reading a frame; that is, does it have a buffer allocated. If not, it allocates a buffer and zeros the bytes read counter.

It does a non-blocking `read()` requesting the a number of bytes given by the frame size minus the number of bytes read. When the `read()` returns, the function increments the bytes read counter by the number of bytes read. If a complete frame has been read (frame size - bytes read $= 0$), the function checks the master channel counter and posts the buffer containing the frame to the sorter exactly as in the other designs.

Whether or not a frame has been completely read, the function returns at this point.

To support this design, there must be a main loop containing a `select()` call which waits for input to be available. (It also waits for outputs to be ready, but more on that later.) When the `select()` triggers on the input, it calls the frame input function outlined above.

3.3 TCP/IP output task

Like the TCP/IP frame input task, there are three flavors of TCP/IP output task, depending on whether native threads are available, and if not, whether ISIS threads are available or we don't have threads at all.

In general, the output task, gets a buffer from a queue and write it to an output file descriptor. Simple.

3.3.1 native threads

The native thread receives as an argument a structure containing a file descriptor to which to write, and a handle for the queue from which it is to get buffers to be output.

The thread calls into the queue to obtain a buffer (which blocks if no data is in the queue) and then executes a blocked `write()` to the given output file descriptor. The `write()` is in a loop to ensure that the whole buffer is written.

Once the buffer has been written, the task loops back to get the next buffer for output.

3.3.2 isis threads

As for the input task, we can get the best performance from the output thread by emulating the native threads design. We replace the blocked `write()` with a non-blocked `write()` coupled with (and following) an `isis_select`. Everything else remains the same.

To more closely emulate the non-threaded design, the output task could be created by `isis_output()`. In this case, everything else is as in the non-threaded design; however, the main loop mentioned is the `isis_mainloop()`.

3.3.3 no threads

If no threads package is available, the output task is a function much like the input task. The argument is the same for this function as for the tasks outlined above.

The output function is called when an output file descriptor is ready to be written. Since we cannot, in general, expect to write a whole buffer in one non-blocked `write()`, the output function may be called several times to write one buffer. To support this, it maintains a static counter of how many bytes of the current buffer it has written.

When the function is entered, it first checks to see if it is in the middle of writing a buffer. If not, it retreives a buffer and zeros the bytes written counter. Note that the output function is not called if there is nothing in the queue to be written, so it cannot block on retreiving a buffer from the queue.

The output does a non-blocking `write()` containing the a number of bytes given by the buffer size minus the number of bytes written. When the `write()` returns, the function decrements the bytes read counter by the number of bytes read.

Whether or not a frame has been completely read, the function returns at this point.

To support this design, there must be a main loop containing a `select()` call which waits for output descritors to be ready for writing. (The same `select()` also waits for inputs to be ready.) When the `select()` triggers on the output, it calls the frame input function outlined above. An output descriptor is added to the `select()`'s write `fd_set` only if there are buffers queued to that output.

3.4 queue object

Like the TCP/IP input and output tasks, there are three flavors of the queue object software, depending on whether native threads are available, and if not, whether ISIS threads are available or we don't have threads at all.

The queue object includes a gatekeeping mechanism which prevents the output from extracting data beyond the end of the queue. This object cooperates in either blocking the output thread execution, if we have a threads package, or arranging for the output function to be called or not, if the code is not multi-threaded.

3.4.1 native threads

When a thread posts a buffer to a queue (usually through a sorter), the queue increments a associated semaphore (calling POSIX `sem_post()`, for example). This operation never blocks.

An output thread associated with a queue calls the queue to obtain the next buffer for output. Before that call retreives a buffer, it issues a `sem_wait()` to decrement the semaphore. This function blocks the thread if the semaphore is zero, indicating that there is no data in the queue.

The thread will be unblocked the next time another thread posts a buffer to the queue. The call will continue, retrieve a buffer, and return it to the output thread (which is then free to block in a `write()`).

3.4.2 isis threads

The design of this aspect of the queue for this case depends on the design of the output task.

3.4.2.1 threaded output

The ISIS threads package doesn't supply a semaphore, as such, but we can construct on easily with the ISIS functions `t_wait()` and `t_sig()` coupled with a variable which counts the number of buffers in the queue.

On posting a buffer, the queue increments the buffer count and issues a `t_sig()` to mark as runnable an output task waiting on the queue.

When retreiving a buffer, the queue checks the number of buffers in the queue. If it's greater than zero, it gets the next buffer, decrements the count, and returns the buffer to the caller. If the buffer count is zero, the queue calls `t_wait()` to block until another buffer is added to the queue.

3.4.2.2 function output

Since in this case the output is called automatically by ISIS as an `isis_output()`, we have to arrange in the queue to remove the output when no buffers are enqueued.

On posting a buffer, the queue calls `isis_output()` to add an output task to write the buffer out of the queue. The queue will need to know what file descriptor is associated with in order to do this.

On retreiving a buffer, the queue calls `isis_wait_cancel()` to remove output task if the queue is empty and it returns an empty queue indication. The output function which recognize the empty queue condition and returns.

3.4.3 no threads

With no threads package, the queue object has to manipulate the write `fd_set` used in the `select()` call in the program's main loop. To do this, the queue object needs to know about the output descriptor with which it is associated.

When posting a buffer, the queue adds the associated output file descriptor to the main loop `select()` call's write `fd_set`. This allows an output function to be called to get the buffer off the queue when the output file is ready for writing.

On retreiving a buffer, the queue removes the the associated output file descriptor from the main loop `select()` call's write `fd_set` if the queue is empty. It returns an empty queue indication to the output function which recognizes this condition and returns.

4 Frame Sorter Design

This is the basic design for the frame sorter I actually propose building for our first effort. It is non-threaded and does not use ISIS.

The `main()` function does a little setup and calls the `mainloop()` function. When the program starts, it configures it's standard input and output as a control connection. It also creates an internet domain socket on which to listen for connection requests.

The `mainloop()` function contains the `select()` system call and dispatch logic: It calls the input, output, and configure functions, depending on what descriptors are ready to be read or written.

The `mainloop()` function is part of a mainloop package, which contains other functions to add and delete inputs and outputs and to add input and output file descriptors to the `select()`.

The designs of the input and output functions have been laid out in the Chapter 3 [Frame & Packet Handling], page 14 chapter.

There also needs to be a function to handle commands to reconfigure the program once it is running. In fact, one way to handle the command line options is to force the user to give an ASCII command rather than traditional UNIX program options.

4.1 main function

4.2 mainloop package

4.3 command handler

The command handler reads ascii commands from a command connection and executes them.

The commands are:

'`req`'	request data from the program
'`go`'	start a data flow
'`cfg`'	reconfigure filtering on a data connection
'`brk`'	break an input or output connection
'`con`'	form a connection to a data source

4.3.1 request data

'`req`' *who what filter*

where:

who is a requester ID string of the form *name:pid*`@`*host* where *name* identifies the process making the request, *pid* is its process ID, and *host* is the hostname where the process is running.

what is either '`tf`' for transfer frames or '`pkt`' for packets. This is included for situations where frames and packets may be handled by the same program.

filter is a string of the form *key,key,...* where each *key* is a virtual channel ID or packet ID. All frames or packets are selected for output by default. If all *key*s are negative, the corresponding frames or packets are filtered out. If any *key*s are positive, the default changes to everything being filtered out and positive key values then select their corresponding frames or packets.

response:

'`ok`' *who ref#*

where:

who is the identifier from the '`req`' command.

ref# is the reference number for the new connection created by the request. This is used in subsequent commands as a handle for the connection.

sorter / assembler command: go ref#

response: (data flows...)

command: cfg who ref# filter

response: ok who ref#

command: brk who ref#

response: ok who ref#

command: con who where where = hostname:port response ok who ref#

5 Ascii Commands

The telemetry software needs to be controlled by the STOL interpreter and other external programs. Toward that end, the various programs will accept ASCII commands and issue ASCII responses.

The following are the commands that will be recognized by the telemetry software. For each, the command actually sent is the two-character abbreviation.

The only external control interface for the telemetry software is the telemetry controller. The controller accepts most of the commands above.

5.1 connect (cn)

The '`cn`' command tells a telemtry `frame_sorter` to connect to a frame source. With the frame parameters given by the '`fp`' command, parameters to '`cn`' give all remaining information needed to ingest the telemetry frames.

Important: This command is accepted only by the `frame_sorter`, not by the telemetry controller.

The syntax of the '`cn`' command is:

```
cn <station> <transport> <source> [<wrapper> ...]
cn break
```

In the first form of the command, the '`cn`' command forms a connection to a telemetry source; and in the second case, it breaks any existing connection. The subsequent explanation deals with the first form.

The *station* parameter is the name of the ground station or device to which we are connecting. It is used only for the formation of archive file names.

The *transport* is the type of network or other transport by which we are to get the telemetry data. Currently, this may be one of:

'`tcp`' for the TCP/IP (transmission control protocol / internet protocol). The `frame_sorter` initiates the connection.

'`server_tcp`'
for the TCP/IP also, but in this case, the `frame_sorter` listens for a connection. If once established the connection is broken, the `frame_sorter` will return to listening for another connection.

'`udp`' for the UDP IP (user datagram protocol / internet protocol).

'`isis`' for the Isis reliable multicasting protocol, which typically is layered upon UDP.

The form of the *source* parameter depends on the *transport*. For '`tcp`' and '`udp`', *source* is a space-separated combination of hostname and IP port number to which to form a TCP connection or a UDP association. For '`server_tcp`', *source* is the port number on which to listen for a connection. If the port number is '`*`', the system will choose an available port. For the '`isis`' transport, *source* is the name of an Isis group from which data is to be obtained.

The optional *wrapper* parameters are names associated with an encapsulation around each frame added by the data source. The wrapper may consist of a header, a trailer, or both. Wrappers should be specified in order, from the outermost to the innermost. Currently recognized wrapper names are:

‘`itp`’ for the 16-byte header that is produced by the Code 521 Front-end Telemetry & Command Processor (FTCP), called the ITP or MEDS header.

‘`ftcp`’ for the combination of the ITP header with an 8-byte trailer also produced by the FTCP.

‘`smexddd`’ for the header derived from combining the DSN (Deep Space Network) Data Delivery (DDD) header with a project-specific secondary header designed for use by TRACE and following SMEX missions.

‘`fep521`’ for the header produced by another Code 521 Front-end Processor, and being used in conjunction with the SPUDD.

‘`anno12t`’ for the 12-byte annotation trailer added to each frame by the `frame_sorter` itself. This must be given as the outermost wrapper when a `frame_sorter` is reading frames from another frame_sorter.

If the data source adds no encapsulation, no *wrapper* should be specified.

The response to a successful ‘`cn`’ command is:

```
ok
```

5.2 acquire (ac)

The ‘`ac`’ command is a request for a telemetry data flow of frames, packets, or data points. It normally is used by a client to request a connection to a source of packets or frames, or by a STOL interpreter to request that a data unpack program be started and connected to a packet source.

The syntax of the acquire command is:

```
ac <transport> [<destination>] <datatype> [<filter>]
ac off <handle>
```

In the first form, a data flow is commanded. The *transport* may be one of:

‘`tcp`’ to request that the telemetry subsystem create a TCP socket and listen for connection requests. The response message will contain the hostname and port number to which the requestor may connect. For this transport, no *destination* may be specified.

‘`udp`’ to request that the telemetry subsystem send the requested data using UDP datagrams. For this transport, the *destination* must be a space-separated hostname and port number combination specifying where the datagrams should be sent.

‘`isis`’ to request that the telemetry subsystem join an Isis group and send the requested data to that group. In this case, the *destination* must be the name of the Isis group to which the data should be sent.

'`client_tcp`'
: to request that the telemetry subsystem initiate a TCP connection to the given *destination*. The *destination* must be a space-separated hostname and port number combination.

Please note that only the the '`tcp`' transport is supported at this time.

The *datatype* argument specifies what kind of data is desired. It may be one of:

'`frames`' for requesting annotated frames. Each frame, complete with it's source header and trailer (if any), will have the standard 12-byte annotation appended to it.

'`pkts`' for requesting annotated packets. This is supported only for CCSDS telemetry. All packets will be prepened with a 16-byte ITP header followed by the standard 12-byte annotation.

'`data`' for requesting data unpacking. The effect of this command is to start a data unpack program and to connect it to a source of packets.

See Section 5.5.1 [filter syntax], page 26 for information on the optional '`filter`' parameter. If a filter is not specified, all data of the requested type will be sent.

Important: Commands requesting packets on multiple virtual channels (for CCSDS telemetry) using either TCP transport presently are not allowed. This is because the virtual channels are demultiplexed and packets are extracted on each channel independently. Applications requiring packets from more than one VC using a TCP transport must make one request and one data connection for each VC on which data is desired. We hope to offer in the near future the ability to multiplex the packets from multiple VCs onto a single connection, and so to make them available using a single request.

The response to a successful '`ac`' command to start a data flow is of the form:

```
ok <handle> [<host> <port>]
```

The *handle* parameter is an opaque (to the client) identifier for the connection which is to be – or has been – establishted. This handle can be passed to '`ac off`' or the filter command '`fi`'.

The *host* and *port* will be included if the requested transport was TCP.

In the second form above, the acquire command stops a data flow.

Termination of data flows by TCP clients should be done simply by breaking the connection. The '`ac off`' command must be used to stop flows using the other transports.

Examples

```
ac tcp pkts vc0 12 1 16 7 22
ok host_a 31009
ac tcp pkts vc2 not 24 27
ok host_b 32012
```

The first command requests packets (application IDs) 1, 7, 12, 16, 22 on virtual channel zero. The response directs us to connect to port 31009 on a machine called "host_a". The second command requests all packets except 24 and 27 on virtual channel two. The response directs us to connect to port 32012 on "host_b" for those packets.

```
ac tcp frames
ok host_a 32110
ac udp myhost 34000 frames not vc2
ok
ac isis mysat_vc0 pkts vc0
ok
```

The first command requests all frames. The response directs us to connect to port 32110 on machine "host_a". The second command asks for all frames except those on virtual channel two to be send in UDP datagrams to port 34000 on host "myhost". The third command directs that all packets on virtual channel zero be sent to the Isis group "mysat_vc0".

(*Please note* that it may be necessary to add a host and port to the response to '`ac udp`' commands when those commands are implemented.)

5.3 archive (ar)

The '`ar`' command tells the telemetry system to store telemetry in disk files as it is ingested and processed.

The syntax of the archive command is:

```
ar <datatype> <filename> <pool_size> [<filter>]
```

The *datatype* parameter specified what type of data should be archived. It may be either '`frames`' to archive annotated frames, or '`pkts`' to archive annotated packets.

The *filename* parameter specifies the name of the archive file. The special name '`default`' may be used to allow the software to generate the filename based on mission and connection parameters.

Archives are composed of one or more fixed size files, and the *pool_size* parameter controls the size of each of those files. If a zero size is given, a default size will be used.

See Section 5.5.1 [filter syntax], page 26 for information on the optional '`filter`' parameter. If a filter is not specified, all data of the requested type will be archived.

The response to a successful archive command is simply '`ok`'.

5.4 playback (pb)

The '`pb`' command is used to request the playback of ITOS telemetry archive files. It is very much like the acquire command, but has three additional parameters.

The syntax of the playback command is:

```
pb <datatype> <transport> <destination> <filename> <timetype> [<start> <stop>] [<filte
pb off
```

The *datatype*, *transport*, and *destination* parameters are exactly as in the aquire command. Note however, that a playback command requesting frames from a file containing packets will fail.

The *filename* parameter specifies the name of the archive to replay. This may be given as an asterisk ('`*`') if actual start and stop times are given.

Archive files may be replayed according to timetags on the data. Both spacecraft and ground-received time is maintained for each transfer frame or packet. The *timetype* parameter tells which of these times to use in comparing against the start and stop times given by the next two parameters. It may be one of:

'`sc`'	to compare against spacecraft time. This is the default.
'`gnd`'	to compare against ground-received time. This is the time at which the frame was received by the ground system.
'`none`'	if no start or stop times are given. In this case, then, the *start* and *stop* must not be included in the command.

The *start* and *stop* parameters are the start and stop times, respectively, over which the archived data should be replayed. If a filename has been specified, one or the other may be given as '`0`' to indicate that all packets before *stop* or all packets from *start* onward should be played from the file. To replay the whole file, use a *timetype* of '`none`'.

See Section 5.5.1 [filter syntax], page 26 for information on the optional '`filter`' parameter. If a filter is not specified, all data of the requested type will be replayed. Note that the same filter restrictions regarding packets on multiple virtual channels exist for playback of frame files as for aquiring live data.

The response to the playback command is identical to that for the acquire command. Section 5.2 [acquire (ac)], page 23

5.5 filter (fi)

The '`fi`' command is used to change the data filtering on an existing connection. For example, if a program is receiving all packets on VC0 and then wishes to exclude packet 10, it can issue a filter command to do it.

The syntax of the filter command is:

```
fi <handle> <filter>
```

The *handle* parameter is a handle returned by an acquire ('`ac`') or playback ('`pb`') command. It is the identifier for the data connection for which we are changing the filtering.

The filter syntax itself is described below.

5.5.1 filter syntax

filter parameters are the means by which the program sending the command tells the telemetry software what data it wants. The *filter* syntax is described by the following grammar:

```
filter:           queue-control filter-op filter-not filter-spec-list
filter-spec-list: /* empty */
                  filter-spec-list filter-spec
filter-spec:      vcid
                  vcid filter-op filter-not apid-list
                  appid-list
apid-list:        /* empty */
```

```
                    apid-list apid
vcid:               'vc0' | 'vc1' | 'vc2' | ... 'vc7'
apid:               <integer 0 through 2047>
filter-op:          /* empty */
                    'add'
                    'drop'
filter-not:         /* empty */
                    'not'
queue-control:      /* empty */
                    'qc' opt-limit <integer queue-pool size>
opt-limit:          /* empty */
                    'limit'
```

Virtual channels are identified by '`vc`' with a trailing digit one through seven; for example, '`vc0`' is virtual channel zero. Packet IDs (also called application IDs) are decimal numbers zero through 2047 (for CCSDS).

If no filter is given, "all" is implied. For packet filters, a VCID not followed by any packet IDs implies all packets on the given VC.

The keyword '`not`' is used to invert the sense of the filter. Normally, the filter sense is positive; that is, you get what you ask for. The '`not`' makes it negative sense: You get everything except what follows the '`not`'.

In the filter ('`fi`') and archive ('`ar`') commands, the keywords '`add`' and '`drop`' may be used in addition to the '`not`' keyword, and in the same locations, to augment existing filtering. This works exactly as one would expect. Requests to drop things not currently active might draw a warning, but are never errors.

Note that a simple list of VCs and/or APIDs and such lists negated (containing '`not`'), form an "absolute" filter; that is, a filter which replaces any filter previously in place for the thing (an output or archive) in question. Filters containing '`add`' and/or '`drop`' keywords modify a previously existing filter.

The queue-control part of the filter may be used to control two aspects of the output queue associated with the given output. The queue-control is introduced with the '`qc`' keyword. If the optional keyword '`limit`' follows, the number of buffers in the output queue will be limited to the size of a single memory pool used by the queue. The size of the queue memory pool is the next argument. Apart from it's use to limit the output queue size, the memory pool size may be used to tune the performance of an output queue.

Examples

```
vc0 vc1 vc2      all packets on VCs 0, 1, & 2.
vc0 21 23 24 vc2 packets 21, 23, & 24 on VC0 & all packets on VC2
1 2 3                   packets 1, 2, & 3
```

The examples above are all in the positive sense; that is, they tell what we want. The '`not`' keyword allows specifications in the negative sense, telling what we don't want:

```
not vc0 vc2 2 13 send all except vc0 & vc2 pkts 2 & 13
vc1 not 2 3 9    send only vc1, but not packets 2, 3, or 9
not vc2 not 2           send all except vc2, but send pkt 2 on vc2 also
```

Note that double negatives are acceptable only in the context given in the third example. The construct "vc0 not vc1" will be rejected as bad syntax.

If we take an output with filtering '`vc0 1 2 3 vc1`':

```
add vc2                 send vc2 in addition
drop vc1                stop sending vc1 data
vc0 add 4 5 6           begin sending pkts 4, 5, & 6 on vc0, also
```

Note that in the last case, the '`add`' could have been placed before the '`vc0`' to the same effect.

```
qc limit 100 vc0        all pkts on vc0, queue no more than 100 pkts
```

Use a queue limit if the process receiving the data can't keep up with the data rate, yet you want the data it sees to be contemporaneous.

5.6 greeting (hi)

The greeting (not really a command) is used to identify a client to the telemetry controller. It is not required of clients, but it allows the controller to call a client by name on controller displays and in event messages.

The syntax of the greeting is:

```
hi <program> <pid> <hostname> [<comment>]
```

Obviously, the client program can supply any information for the parameters, but the controller expects the following.

The *program* parameter is the name of the client program; *pid* is the client's process ID; and *hostname* is the name of the machine where the client program is running.

The *comment* is a short, optional explanatory text.

There currently is no response to the greeting, though I suppose the controller should respond with '`hi`', just to be polite.

5.7 frmparms (fp)

The '`fp`' command is used to tell the `frame_sorter` what sort of frames it will be processing. This command should not be sent to the telemetry controller. It is sent from the controller to frame_sorters, with the appropriate parameters derived from the mission database.

The syntax of the frameparms command is:

```
fp <mission> <frame_type> [<tf_length> <tf_version> <scid>]
```

The *mission* parameter is used to form the default archive file name. It should be the mission name.

The *frame_type* is the type of frame the `frame_sorter` can expect to read. Presently, '`ccsds`' is the only frame type supported. In the near future, we expect to add a type to support Spartan missions.

For CCSDS frames, three additional parameters are required:

The *tf_length* is the CCSDS transfer frame length, including the sync and CRC bytes. This length does not include any header or trailer added by the telemetry source (a ground station or front-end processor).

The *tf_version* is the expected CCSDS transfer frame version. Presently, only version 1 frames are supported, so this always should be set to zero.

The *scid* parameter is the spacecraft the `frame_sorter` should expect to find in the transfer frame header.

The standard '`ok`' response is given for successful frameparms command.

5.8 reasm (ra)

This command is issued by the controller to the `frame_sorter` to begin (or stop) reassembling CCSDS source packets on one virtual channel.

The syntax of the reassemble command is:

```
ra <vc> [stop]
```

The *vc* parameter is a single digit zero to seven indicating the virtual channel on which packets should be reassembled. Appending the keyword '`off`' changes the '`ra`' into a command to stop reassembling packets on the given virtual channel.

The response to this command depends on the current state of the frame sorter. For commands to start reassembly, if the input is connected, the response will be '`ok start`'; otherwise, it will be '`ok setup`'.

For commands to stop reassembly, the response is '`ok stop`'. This is true even if we are not reassembling packets on the given VC.

Index

(Index is nonexistent)

Table of Contents